Formalizing Traceability and Derivability in Software Product Lines

Swarup Mohalik, Ramesh S., Jean-vivien Millo
India Science Lab
General Motors, TCI
Bangalore, India
Email: {swarup.mohalik,ramesh.s,jean.v}@gm.com

Shankara Narayanan Krishna, Ganesh Narwane
Dept. of Computer Science and Engineering
I.I.T., Powai
Mumbai, India
{krishnas,ganeshk}@cse.iitb.ac.in

***Abstract*—In the literature, the definition of product in a Software Product Line (SPL) is based upon the notion of consistency of the constraints, imposed by variability and traceability relations on the elements of the SPL. In this paper, we contend that consistency does not model the natural semantics of the implementability relation between problem and solution spaces correctly. Therefore, we define when a feature can be *derived* from a set of components . Using this, we define a product of the SPL by a ⟨specification, architecture⟩ pair, where all the features in the specification are derived from the components in the architecture. This notion of derivability is formulated in a simple yet expressive, abstract model of a productline with traceability relation. We then define a set of SPL analysis problems and show that these problems can be encoded as Quantified Boolean Formulas. Then, QSAT solvers like QUBE can be used to solve the analysis problems. We illustrate the methodology on a small fragment of a realistic productline.**

***Keywords*-Software Product Line; Sanity analysis; Formal methods; QSAT**

I. Introduction

Software Product Line (SPL) is a development framework to jointly design a family of closely related software *products* in an efficient and cost-effective manner. Every SPL is built upon a collection of features and components. Each individual product is *specified* by a subset of features

Each product in the family is *specified* by a set of features drawn from a collection common to the family, and is *implemented* by an architecture comprising a set of reusable components selected from a collection of *basic assets* which are developed once for the entire family.

There are two key orthogonal aspects of an SPL, namely, *variability* and *traceability*. While variability introduces different choices (termed *variation points*) within the artifacts in system development, such as specifications, architectures and components, traceability relates the variation points together across the artifacts. Since variability introduces complex constraints among the variation points, managing variability in large industrial SPLs is quite complex and has given rise to a number of analysis problems. These have been the focus of SPL research in the recent years. A comprehensive survey of these analysis problems and their solutions can be found in Benavides et al.[1].

On the other hand, we observe that traceability and its implications have not been studied in as much depth in the literature. In the following, we mention the few works addressing traceability as a primary aspect. It is defined in [2] as one of the four important characteristics of a variability model, namely, consistency, visualization, scalability and traceability. A variability management model that focuses on the traceability aspect between the notion of problem and solution spaces is presented in [3]. Anquetil et al.[4] formalize the traceability relations across problem and solution space and also across domain and product engineering. In [5], the notion of *product maps* is defined which is a matrix giving the relation between features and products. Consistency analysis of product maps is presented in [6]. Zhu et al.[7] define a traceability relation from requirement to feature and also from feature to architecture with consistency analysis. [8] presents a consistency verification method between feature model and architecture model. Metzger et al.[9] differentiate SPL variability and product variability and then present a framework based on OVM by Pohl et al.[10] to perform checks for consistency, liveness, commonness, realizability (completeness), and flexibility (soundness).

One of the central concepts of the SPL analyses in the above-mentioned works is that of a *product*. It is defined through the notion of consistency between a collection of features and components and the constraints imposed by variability and traceability. In this report, we contend that consistency does not model the natural semantics of the implementability relation between problem and solution spaces correctly. It allows components and features to co-exist without any conflict, but it also allows cases where the features may not be derivable from the components. Hence, the SPLs can be shown to allow products where the components are not related to the features in a more intuitive notion of traceability. Therefore, we define when a feature can be *derived* from a set of components. Using this, we define a product of the SPL by a ⟨specification, architecture⟩ pair, where all the features in the specification are derived from the components in the architecture. This definition of products is tighter than the existing "consistency" based definitions.

Another contribution of the report is a simple yet expressive, abstract model of a productline where we formally define the derivability notion through the traceability relation. We then define a set of SPL analysis problems. Some of these problems are already addressed in earlier works but are redefined in the light of the new concepts. The others are new and arose because of the separation of problem and solution space linked through traceability. We show that these problems, in general, can be encoded as Quantified Boolean Formulas(QBF) and QSAT solvers[11] can be used to solve the problems. We illustrate the methodology on a small fragment of a realistic productline.

The summary of our contributions in this report are the following:

1) A new definition for SPL products based on a notion of derivability of feature specifications from component architectures. The traceability relation plays the central role in this definition.
2) A simple, abstract semantic model of SPL with traceability. The model abstracts out the details from the existing descriptions of SPL in the literature and allows us to define the core concepts in a formal and concise manner.
3) A set of analysis problems in the SPL, some of which are known but cast anew in the light of the new definitions, and others that are novel.
4) A solution method for the analysis problems which is based on QBF encoding and QSAT solving. This is necessitated by the nature of some of the analysis problems and is in contrast to the SAT based solving methods generally employed for the extant SPL analyses.

Outline of The Report: In the following section, we introduce a case study of Entry Control Product Line (ECPL) from the automotive domain. This is used as a running example throughout the rest of the report. The formal model of an SPL with traceability is described in Section III. It introduces the central notion of derivability and the analyses we would like to carry out in SPL. In Section V, we show how the analysis problems can be encoded in QBF. The results of the analyses using QSAT on the ECPL case study is presented in Section IV. Finally, we conclude in Section VIII with a summary of the report and some future directions. The proof of the main result relating the analysis problems and QBF formulae is given in the appendix.

II. The Entry Control Product Line (ECPL)

We introduce a fragment of a typical Entry Control Product Line (ECPL) used in the automotive industry. It will be used to illustrate the concepts throughout the report and as a case study in Section IV. The entry control system comprises all the features involved in the controlling of door locking/unlocking in a car. In this study, we focus on the following subset:

- Manual lock: controls the locking/unlocking through manual lever presses
- Power lock: controls the locking/unlocking according to key button press, courtesy switch press and sill button press.
- Door lock: controls automatic locking of doors when the vehicle starts.
- Door relock: controls automatic relocking of doors in case of pick up/drop and drive.

The ECPL feature diagram: Figure 1 presents the feature diagram of the ECPL (a la Czarnecki [12]). The dark gray boxes are features of the ECPL. The light gray boxes are parameters modeled as features. The Power lock feature is mandatory. Manual lock is optional. When it is present, the Power lock feature is excluded. The Door lock feature is optional and can be triggered either when gear is shifted out of park or when car speed reaches a predefined value. The Door relock feature is optional. The car should have either a manual or an automatic transmission. Manual transmission disallows the "park options" of Door lock since there is no park gear in a manual gearbox.

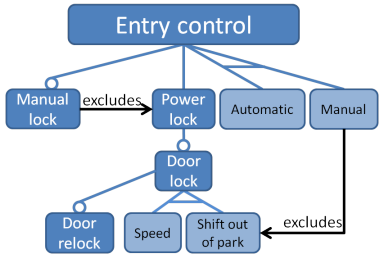

Figure 1. The feature diagram of the ECPL.

The ECPL architectural diagram: Figure 2 represents the platform of ECPL using a notation called Modal Architectural Model (abbreviated as MAM). It is a simplified form of EASEL by [13] and yet preserves the essential notion of variability central to the product line. The platform is composed of three components: *Door lock manager*, *Power lock*, and *Auto lock*. The first is mandatory but the two others are optional (denoted by dotted boxes). The system has seven "in" ports (dark squares) and three "out" ports (light squares). The interconnections between external and internal ports connect ports of the same type but internal interconnection connect complementary ports ("out" port to "in" port). The signals "Transmission in Park" and "Speed" are alternatives. Similarly "Automatic" and "Manual" inform the system on the type of transmission.

Auto lock component requires two global input signals while *Power lock* component requires five. They provide lock/unlock command signals to *Door lock manager*. The command provided by *Power lock* component depends upon manual action, and the command provided by *Auto lock* component is according to the requirements of the

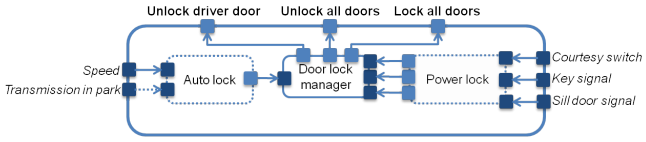

Figure 2. The platform of the ECPL.

features Door lock and Door relock.

The *Door lock manager* component arbitrates the lock/unlock command signals from *Auto lock* and *Power lock* and forwards them to the global outputs depending upon a calibration (1/Unlock all doors, 2/Unlock Driver door, 3/Lock all doors).

The traceability relations of the ECPL: To avoid confusion between the homonymous features and components (*Automatic*, *Manual*, and *Speed*), we will, in the sequel, prefix the labels with $f\_$ or $c\_$ respectively. Table I presents the required components to implement each feature.

Feature	Component
Power lock	Door lock manager& Power lock
Door lock	Auto lock
Door relock	Auto lock
f_Automatic	c_Automatic
f_Manual	c_Manual
Shift out of Park	Gear in Park
f_Speed	c_Speed

Table I
EACH FEATURE REQUIRES COMPONENT(S)

Table II presents the features provided by the architectural elements.

Component/Interconnection	Feature
Door lock manager & Power lock	Power lock
c_Automatic	f_Automatic
c_Manual	f_Manual
Auto lock	Door lock & Door relock
Gear in park	Shift out of park
c_Speed	f_Speed

Table II
THE ARCHITECTURAL ELEMENTS PROVIDE SOME FEATURES

III. MODEL OF SPL : TRACEABILITY AND IMPLEMENTATION

In this section, we propose a model of the software productline making the traceability relation explicit and define an implementation relation between architectures and specifications based on traceability.

A. Modeling Decisions

In [9], the traceability relation is given as a set of arbitrary propositional constraints over the components and features. In the current report, we impose a fairly natural structure on the traceability relation, consisting of a *provides* and a *requires* function for each feature. This is inspired by the points of view of the suppliers and integrators (OEMs). Suppliers usually would package one or more features in a component, which is captured by the provides relation. On the other hand, integrators start with a set of features which requires a set of components for implementation.

Importantly, the implementations are related to the specifications only when they can be derived using the traceability relation. Consider a simple SPL consisting of a feature f and a component c, but without any traceability relation between f and c. According to analyses such as in [9], since $\{f, c\}$ is consistent (in a propositional logic), it is considered as a product. Clearly, it is not natural. On the other hand, if f was provided by c, then $\{f, c\}$ would be a natural product.

Another novel point in our model is the notion of approximate implementation (Covers). In the literature, the definition of implementation is usually exact: we need the components that provide exactly the same set of features in a specification. However, since many components are prebuilt by the suppliers, there may not be a choice suitable for an exact implementation. For example, if the OEM wants a feature of ABS (Anti-lock Braking) and the supplier has packaged both ABS and TC(Traction Control) in one component, the OEM has to choose this component which covers (but does not exactly implement) the specification of ABS.

B. Formal Model

Let $\mathcal{F}$ be a set of features. A subset of $\mathcal{F}$ is called a *specification*. The scope of an SPL is a collection of specifications: $\overline{\mathcal{F}} \subseteq \wp(\mathcal{F})$. The specifications are implemented using a set of (reusable) components $\mathcal{C}$. Each subset of $\mathcal{C}$ is called an *architecture*. An SPL platform consists of a set of architectures: $\overline{\mathcal{C}} \subseteq \wp(\mathcal{C})$.[1]

A traceability relation $\mathcal{T}$ connects the features and components: $\mathcal{T}$ is specified as a pair $\langle prov, req \rangle$ where $prov$ and req are maps $\mathcal{F} \rightarrow \wp(\wp(\mathcal{C}))$. Through the traceability relation we capture the *sufficient* ($prov(.)$) and *necessary* ($req(.)$) conditions to implement a feature. When $prov(f) = \{C_1, C_2\}$, we interpret it as the fact that the set of components C_1 (also, C_2) provides the implementation of the feature f. On the other hand, when $req(f) = \{D_1, D_2\}$, we interpret as the fact that the implementation of the feature f requires the set of components D_1 or the set of components D_2.

Definition 1. *An SPL Ψ is defined as a triple $\langle \overline{\mathcal{F}}, \overline{\mathcal{C}}, \mathcal{T} \rangle$, where $\overline{\mathcal{F}}$ is the scope, $\overline{\mathcal{C}}$ is the platform and $\mathcal{T}$ is the traceability relation.*

[1]The representation of specification and platform is semantic in nature. Syntactic representation of these may use FODA diagrams, MaMs or a variety of notations in the literature. In general, one can have implicit representations through constraints on the features and components; we will adopt this view in the following sections.

In the ECPL case study, $\mathcal{F}$ contains the nine features of Figure 1 and the ECPL scope $\overline{\mathcal{F}}$ contains eight specifications. For illustration, we choose the following specifications: $spec_1 = \{Power\,lock, f\_Automatic\}$ and $spec_2 = \{Power\,lock, f\_Automatic, Door\,lock, Shift\,out\,of\,park, Door\,relock\}$. The top-most feature *Entry control* is in every specification and is not mentioned explicitly.

In ECPL, $\mathcal{C}$ contains the three components of Figure 2 and the twelve interconnections which are also modeled as components. Note that the mandatory interconnections are in every architecture and are not mentioned explicitly. The ECPL platform $\overline{\mathcal{C}}$ contains nine architectures which can be extracted from the ECPL platform. Again, for illustration, we select two architectures $arch_1 = \{Door\,lock\,manager\}$ or $arch_2 = \{Door\,lock\,manager, Power\,lock, c\_Automatic, Auto\,lock, Transmission\,in\,park\}$.

The traceability relation in ECPL is given through the Tables I($req(.)$) and II($prov(.)$). For example, the $Auto\,lock$ component provides the features $Door\,lock$ and $Door\,relock$. Each of these features requires only $Auto\,lock$ component.

The main concept of implementability in Ψ is defined as follows: a feature is implemented by an architecture (set of components in $\overline{\mathcal{C}}$) if the architecture provides the feature and simultaneously fulfills the mandatory requirements of the feature.

Definition 2 (Implements)**.** *Given an SPL* $\Psi = \langle \overline{\mathcal{F}}, \overline{\mathcal{C}}, \mathcal{T} \rangle$, $implements_{\Psi}(C, f)$ if $\exists C_1 \in prov(f), C_2 \in req(f) \cdot C_2 \subseteq C_1 \subseteq C$.

The set of features implemented by an architecture C *is defined as* $Provided\_by_{\Psi}(C) = \{f | implements_{\Psi}(C, f)\}$.

In ECPL, $implements_{\Psi}(spec_2, Power\,lock)$ holds but $implements_{\Psi}(spec_1, Power\,lock)$ does not hold. Moreover, if one considers $prov$ as given in Table II without the last line, $implements_{\Psi}(arch, f\_Speed)$ never holds for any architecture $arch$ because $prov(f\_Speed) = \emptyset$ even if $req(f\_Speed) = \{\{c\_Speed\}\}$.

With the basic definitions above, we can now define when an architecture exactly implements a specification.

Definition 3 (Realization)**.** *Given* $C \in \overline{\mathcal{C}}$ *and* $F \in \overline{\mathcal{F}}$, $Realizes(C, F)$ *if* $F = Provided\_by(C)$.

Due to the required equality, we have the following easy result.

Proposition 4. *An architecture realizes at most one specification in an SPL.*

The $realizes$ definition in the above imposes a strictness on the implementations. Thus, in the ECPL example, the architecture $arch_2$ realizes the specification $spec_2$, but it does not realize $spec_1$ even though it provides the implementation of all the features of $spec_1$. In many cases, this may be a practical definition. Hence, we relax the definition of realization in the following.

Definition 5 (Covers)**.** *Given* $C \in \overline{\mathcal{C}}$ *and* $F \in \overline{\mathcal{F}}$, C *covers* F *if* $Provided\_by(C) \in \overline{\mathcal{F}} \wedge F \subseteq Provided\_by(C)$.

The additional condition ($Provided\_by(C) \in \overline{\mathcal{F}}$) is added to ensure that the chosen C provides the implementation of a specification in the scope. In ECPL, $\mathcal{C}_2$ covers $\mathcal{F}_1$ but $\mathcal{C}_1$ does not cover (or even realize) anything.

Given $F, F' \in \overline{\mathcal{F}}$, let $F \subset F'$, Then, F' is called the extension of F. The following simple proposition establishes a connection between the relations *realizes* and *covers*. Figure 3 depicts these relations pictorially.

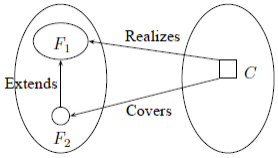

Figure 3. Specification F_1 extends F_2, Architecture C realizes F_1 and covers F_2

Proposition 6. *Given* $C \in \overline{\mathcal{C}}$ *and* $F \in \overline{\mathcal{F}}$ *and* C *covers* F. *Then, there is an extension* F' *of* F *in* $\overline{\mathcal{F}}$ *such that* $Realizes(C, F')$. *Hence, if there is no extension of* F *in* $\overline{\mathcal{F}}$, *then* $Realizes(C, F)$.

In the ECPL case study, $arch_2$ covers $spec_1$, $spec_2$ extends $spec_1$, and $arch_2$ realizes $spec_2$.

The set of products of the SPL are now defined as the specifications and the architectures implementing them through the traceability relation.

Definition 7 (SPL Products)**.** *Given an SPL* $\Psi = \langle \overline{\mathcal{F}}, \overline{\mathcal{C}}, \mathcal{T} \rangle$, *the products of the SPL denoted as* $Prod(\Psi) \equiv \{\langle F, C \rangle | Covers(F, C), F \in \overline{\mathcal{F}}, C \in \overline{\mathcal{C}}\}$

In the ECPL, out of 8 specifications and 9 architectures, there are 11 products. Even if the architecture $arch_3 = \{Door\,lock\,manager, Power\,lock, c\_Manual, Auto\,lock, Transmission\,in\,park\}$ "covers" the specification $\{Power\,lock, f\_Manual\}$, this pair is not a product because $Provided\_by(arch_3)$ is not in the scope $\overline{\mathcal{F}}$. This is because $arch_3$ provides features f_Manual and $Shift\,out\,of\,park$ which should be exclusive.

C. SPL Level Properties

Given an SPL $\langle \overline{\mathcal{F}}, \overline{\mathcal{C}}, \mathcal{T} \rangle$, we define two important relationships between the scope (specification space) and platform (architecture, or implementation, space).

1) Completeness: An SPL $\langle \overline{\mathcal{F}}, \overline{\mathcal{C}}, \mathcal{T} \rangle$ is complete if $\forall F \in \overline{\mathcal{F}} \cdot \exists C \in \overline{\mathcal{C}} \cdot Covers(C, F)$.

The completeness property of the SPL determines if the platform for the SPL is adequate to provide implementation for all the specifications in its scope.

The ECPL is complete. For illustration's sake, let us omit the last entry in Table II. Then, none of the specifications which include the feature f_Speed is realizable because f_Speed cannot be derived from any component.

2) Soundness: An SPL $\langle \overline{\mathcal{F}}, \overline{\mathcal{C}}, \mathcal{T} \rangle$ is sound if $\forall C \in \overline{\mathcal{C}} \cdot \exists F \in \overline{\mathcal{F}} \cdot Covers(C, F)$.

The soundness property relates to the non-redundancy of the platform in an SPL. If the architectures (sets of components) are generated using certain rules or constraints, soundness stipulates that only those architectures which provide an implementation of some specification are generated.

The ECPL is not sound because, for example, the architecture $arch_1$ does not realize any specification (feature set). This is the case with all the architecture where $Power\,lock$ is absent. Now, let us assume that the component $Power\,lock$ is mandatory. The ECPL is still not sound because of $arch_3$ only. If $arch_3$ is omitted from the platform, the remaining ECPL become sound.

3) Existentially Explicit: Given an SPL, and a specification $F \in \overline{\mathcal{F}}$, it is called an existentially explicit specification in the SPL if there exists a $C \in \overline{\mathcal{C}} \cdot Realizes(C, F)$.

In ECPL, $spec_1$ and $spec_2$ are existentially explicit. However, another specification $spec_3 = \langle Power\,lock, f\_Automatic, Door\,lock, Shift\,out\,of\,park \rangle$ is not, because none of the architecture realizes a specification with $Door\,lock$ and without $Door\,relock$.

4) Universally Explicit: Given an SPL, and a specification $F \in \overline{\mathcal{F}}$, it is called a universally explicit specification in the SPL if (i) there exists a $C \in \overline{\mathcal{C}} \cdot Realizes(C, F)$ and (ii) for all $C \in \overline{\mathcal{C}} \cdot Covers(C, F) \Rightarrow Realizes(C, F)$.

In ECPL, $spec_2$ is universally explicit. $spec_1$ is existentially explicit but not universally explicit because it is covered but not realized by the $arch_2$.

It follows from Proposition 6 that

Proposition 8. *If $F \in \overline{\mathcal{F}}$ is covered by some architecture but is not extendable, then it is universally explicit. If F is universally explicit, then none of its extensions has a covering architecture.*

In the ECPL, $spec_2$ is covered and cannot be extended; so it is universally explicit. On the contrary, if a specification has an extension which is covered, the same also covers the extended specification.

5) Unique Implementation: A given specification may be implemented by multiple architectures. This may be a desirable criterion of the platform from the perspective of optimization among various choices. Thus the specifications which are implemented by single architectures are to be identified.

$F \in \overline{\mathcal{F}}$ has a unique implementation if $\exists C \in \overline{\mathcal{C}} \cdot (Covers(C, F) \wedge \forall C' \in \overline{\mathcal{C}} \cdot (Covers(C', F) \Rightarrow C = C'))$.

In ECPL, each specifications including $Door\,relock$ has a unique implementation. On contrary, $spec_1$ has more than one implementation.

6) Common, live and dead elements: Identification of common, live and dead elements in an SPL is one of the basic analyses identified in the SPL community. We redefine these concepts in terms of the our notion of products.

1) An element e is *common* if $\forall \langle F, C \rangle \in Prod(\Psi) \cdot e \in F \cup C$.
2) An element e is *live* if $\exists \langle F, C \rangle \in Prod(\Psi) \cdot e \in F \cup C$.
3) An element e is *dead* if $\forall \langle F, C \rangle \in Prod(\Psi) \cdot e \notin F \cup C$.

In ECPL, the feature *Manual lock* is dead. All the other features are live. The component *Door lock manager* is common.

7) Superfluous Component: A component is superfluous if the platform without the component suffices to provide the same set of specifications.

Let $P \subseteq Prod(\Psi)$, $spec(P) = \{F | \langle F, C \rangle \in P\}$. Let $Prod_{\neg c}(\Psi) = \{\langle F, C \rangle | \langle F, C \rangle \in Prod(\Psi) \wedge (c \notin C)\}$. c is Superfluous if $spec(Prod(\Psi)) = spec(Prod_{\neg c}(\Psi))$.

Superfluousness is relative to a given platform. If in an SPL Ψ, $prov(f) = \{\{a\}, \{b\}\}$, $\overline{\mathcal{F}} = \{\{f\}\}$ and $\overline{\mathcal{C}} = \{\{a\}, \{b\}\}$, then both a and b are superfluous w.r.t. Ψ, whereas if either $\{a\}$ or $\{b\}$ is removed from the platform, the remaining $\{b\}$ or $\{a\}$ is not superfluous anymore (w.r.t. the reduced SPL).

Lemma 9. *Let $c \in \mathcal{C}$ be Superfluous for Ψ. Then, for every $C \in \overline{\mathcal{C}} (c \in C \Rightarrow (\exists C' \in \overline{\mathcal{C}} \cdot c \notin C' \wedge Provided\_by(C) = Provided\_by(C'))$.*

8) Redundant Component: A component is redundant if it is not contributing to any feature in any architecture in the platform. $c \in \mathcal{C}$ is redundant if for every $C \in \overline{\mathcal{C}} (c \in C \Rightarrow (\exists C' \in \overline{\mathcal{C}} \cdot (c \notin C' \wedge C' \subseteq C \wedge Provided\_by(C) = Provided\_by(C'))$.

Note that redundancy is a stronger version of superfluousness; a redundant component is superfluous whereas a superfluous element many not be redundant.

In ECPL, no component is neither superfluous nor redundant. Let us assume that we have a component called $Door\,Relock_{Alt}$ such that $\{Door\,Relock_{Alt}, Auto\,lock\}$ provides the feature $Door\,Relock$. This component would be redundant because $Auto\,lock$ already provides the feature $Door\,Relock$.

It is expected that an SPL can be optimized by omitting the redundant components without affecting the set of products.

Lemma 10. *Let $c \in \mathcal{C}$ be redundant. Construct a SPL $\Psi' = \langle \overline{\mathcal{F}}, \mathcal{T}', \overline{\mathcal{C}} \rangle$ where, $\mathcal{T}'$ be a traceability relation with $req'(f) = req(f) \setminus \{C | c \in C\}$ and $prov'(f) = prov(f) \setminus \{C | c \in C\}$. Then, $Prod(\Psi) = Prod(\Psi')$.*

9) Critical Component: Given an $f \in \mathcal{F}$, a component c is critical for f if for all $C \in \overline{\mathcal{C}}, (c \notin C \Rightarrow \neg implements_{\Psi}(C, f))$.

In ECPL, all the components are critical. Let us assume a component $Auto\,lock_{Alt}$ which is an alternative to $Auto\,lock$ and also provides the feature $Door\,lock$. In such case neither $Auto\,lock$ or $Auto\,lock_{Alt}$ are critical for the feature $Door\,lock$ but $Auto\,lock$ remains critical for the feature $Auto\,relock$.

10) Emerging Features: When a specification is not realizable, but is covered by one or more architectures, the emerging features $Emerging(F) \equiv \{\langle C, Provided\_by(C) \setminus F\rangle | Covers(C, F)\}$.

$Emerging(F)$ gives the covering architectures and the emerging features corresponding to the architecture.

In ECPL, while considering the only architecture that cover $\langle Power\,lock, Manual, Door\,lock, f\_Speed\rangle$, $Door\,relock$ will emerge.

D. Canonical Traceability Relation

A given traceability relation can be reduced to a canonical form without affecting the set of features implementable in the SPL. We define the canonical form in the following.

Definition 11. *$\mathcal{T}$ is* non-redundant *if for every feature f,*

1) *$C_i, C_j \in prov(f), i \neq j$ implies $C_i \not\subseteq C_j$, and*
2) *$C_i, C_j \in req(f), i \neq j$ implies $C_i \not\subseteq C_j$.*

Intuitively, if a smaller set of components implements a feature, a larger set also will. On the other hand, if a larger set of components is required to implement a feature, a smaller set is required automatically. Given a traceability relation, one can check if it is non-redundant and convert it to a non-redundant relation by removing the larger (resp. smaller) sets in $prov(f)$ (resp. $req(f)$).

Definition 12. *$\mathcal{T}$ is* internally consistent *if $\forall f \in \mathcal{F}$, $\forall C \subseteq \mathcal{C}$, $(C \in prov(f) \Rightarrow (\exists C' \in req(f) \cdot C' \subseteq C))$.*

Intuitively, internal consistency of a traceability relation states that each set of components in $prov(f)$ can indeed satisfy the mandatory requirements (coming from $req(f)$ of f.

Given a traceability relation, we can reduce it to a canonical form by the following operations for the $prov(.)$ and $req(.)$ of each feature f.

Claim 13. *For a given SPL $\Psi = \langle \overline{\mathcal{F}}, \overline{\mathcal{C}}, \mathcal{T}\rangle$, the above procedure results in a canonical traceability relation $\mathcal{T}'$ such that for all $C \subseteq \mathcal{C}$, $implements_{\Psi}(C, f)$ iff $implements_{\Psi'}(C, f)$.*

Proof: The canonization algorithm stops when no rules are applicable. Then the conditions of the rules ensure that the resulting traceability relation is canonical.

In order to prove the preservation of implementability, it is easy to show that each rule preserves implementability. ■

Theorem 14. *If Ψ is an SPL with a canonical traceability relation, $implements_{\Psi}(C, f)$ if $\exists C_1 \in prov(f) \cdot C_1 \subseteq C$.*

Algorithm 1 Canonization of Traceability Relation

1: **if** $prov(f) = \emptyset$ or $prov(f)$ is undefined **then**
2: $prov(f) \leftarrow \perp; req(f) \leftarrow \perp$
3: **end if**
4: **if** $C_i, C_j \in prov(f), i \neq j$, $C_i \subseteq C_j$ **then**
5: $prov(f) \leftarrow prov(f) \setminus \{C_j\}$.
6: **end if**
7: **if** $C_i, C_j \in req(f), i \neq j$, $C_i \subseteq C_j$ **then**
8: $req(f) \leftarrow req(f) \setminus \{C_i\}$.
9: **end if**
10: **if** $C \in prov(f)$, but $forall C_i \in req(f), C_i \not\subseteq C$ **then**
11: $prov(f) \leftarrow prov(f) \setminus \{C\}$.
12: **end if**

Short-Hand	Feature
F_1	$Manual\ Lock$
F_2	$Power\ Lock$
F_3	$Door\ Lock$
F_4	$Door\ Relock$
F_5	$F\_automatic$
F_6	F_manual
F_7	F_speed
F_8	$Shift\ out\ of\ Park$

Table III
FEATURES IN ECPL.

Proof: In a canonical traceability relation, due to internal consistency, for every $C' \in prov(f), \exists C'' \in req(f) \cdot C'' \subseteq C'$. Hence the result. ■

Since one can always canonize the traceability relation of an SPL, henceforth we will assume that the SPL under scope is canonical. Thereby, the definition of implementation will henceforth be as given in 14.

IV. Analysis of the ECPL

In this section, we analyze some properties of the ECPL example using QuBE.

In ECPL, there are total 8 Features and 13 Components. The features are listed in Table III and the components are given in Table IV.

A specification is a subset of Features $\mathcal{F}$. The scope of an SPL is a collection of specifications: $\overline{\mathcal{F}} \subseteq \wp(\mathcal{F})$. In our example, scope of ECPL is $\overline{\mathcal{F}} = \{ S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8\}$. All the specifications are represented in tabular form as shown in Table V. A specification corresponds to a column and the 1's in the column select the features in the specification.

1) $S_1 = \{Power\ Lock, F\_automatic\}$
2) $S_2 = \{Power\ Lock, F\_manual\}$
3) $S_3 = \{Power\ Lock, F\_automatic, Door\ Lock, F\_speed\}$
4) $S_4 = \{Power\ Lock, F\_manual, Door\ Lock, F\_speed\}$

Short-Hand	Component
C_1	*Door Lock Manager*
C_2	*Unlock Driver Door*
C_3	*Unlock all doors*
C_4	*Lock all doors*
C_5	*Auto Lock*
C_6	*Power Lock*
C_7	*Courtesy switch*
C_8	*Key signal*
C_9	*Sill door signal*
C_{10}	*C_automatic*
C_{11}	*C_manual*
C_{12}	*Gear in park*
C_{13}	*C_speed*

Table IV
COMPONENTS IN ECPL.

5) S_5 = {*Power Lock*, *F_automatic*, *Door Lock*, *Shift out of Park*}
6) S_6 = {*Power Lock*, *F_automatic*, *Door Lock*, *F_speed*, *Door relock*}.
7) S_7 = {*Power Lock*, *F_manual*, *Door Lock*, *F_speed*, *Door relock*}.
8) S_8 = {*Power Lock*, *F_automatic*, *Door Lock*, *Shift out of Park*, *Door relock*}.

An architecture is a subset of components $\mathcal{C}$. An SPL platform consists of a set of architectures: $\overline{\mathcal{C}} \subseteq \wp(\mathcal{C})$. In ECPL, the platform is $\overline{\mathcal{C}} = \{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9\}$. The architectures are represented in Table VI.

1) A_1 = {*Door Lock Manager*, *Unlock Driver Door*, *Unlock all doors*, *Lock all doors*}
2) A_2 = {*Door Lock Manager*, *Unlock Driver Door*, *Unlock all doors*, *Lock all doors*, *Auto Lock*, *C_speed*}
3) A_3 = {*Door Lock Manager*, *Unlock Driver Door*, *Unlock all doors*, *Lock all doors*, *Auto Lock*, *Gear in park*}
4) A_4 = {*Door Lock Manager*, *Unlock Driver Door*, *Unlock all doors*, *Lock all doors*, *Power Lock*, *Courtesy switch*, *Key signal*, *Sill door signal*, *C_automatic*}
5) A_5 = {*Door Lock Manager*, *Unlock Driver Door*, *Unlock all doors*, *Lock all doors*, *Power Lock*, *Courtesy switch*, *Key signal*, *Sill door signal*, *C_manual*}
6) A_6 = {*Door Lock Manager*, *Unlock Driver Door*, *Unlock all doors*, *Lock all doors*, *Auto Lock*, *C_speed*, *Power Lock*, *Courtesy switch*, *Key signal*, *Sill door signal*, *C_automatic*}
7) A_7 = {*Door Lock Manager*, *Unlock Driver Door*, *Unlock all doors*, *Lock all doors*, *Auto Lock*, *C_speed*, *Power Lock*, *Courtesy switch*, *Key signal*, *Sill door signal*, *C_manual*}
8) A_8 = {*Door Lock Manager*, *Unlock Driver Door*, *Unlock all doors*, *Lock all doors*, *Auto Lock*, *Gear in park*, *Power Lock*, *Courtesy switch*, *Key signal*, *Sill door signal*, *C_automatic*}
9) A_9 = {*Door Lock Manager*, *Unlock Driver Door*, *Unlock all doors*, *Lock all doors*, *Auto Lock*, *Gear in park*, *Power Lock*, *Courtesy switch*, *Key signal*, *Sill door signal*, *C_manual*}

Specifications / Features	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
F_1								
F_2	1	1	1	1	1	1	1	1
F_3			1	1	1	1	1	1
F_4						1	1	1
F_5	1		1		1			
F_6		1		1		1		1
F_7			1	1		1	1	
F_8					1			1

Table V
SPECIFICATIONS IN TABULAR FORM.

Architectures / Components	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9
C_1	1	1	1	1	1	1	1	1	1
C_2	1	1	1	1	1	1	1	1	1
C_3	1	1	1	1	1	1	1	1	1
C_4	1	1	1	1	1	1	1	1	1
C_5		1	1			1	1	1	1
C_6				1	1	1	1	1	1
C_7				1	1	1	1	1	1
C_8				1	1	1	1	1	1
C_9				1	1	1	1	1	1
C_{10}				1		1		1	
C_{11}					1		1		1
C_{12}			1					1	1
C_{13}		1				1	1		

Table VI
ARCHITECTURES IN TABULAR FORM.

The traceability relations (provides and requires) are as in Tables I and II. We reproduce the tables here for ease of reference.

Feature	Component
Power lock	Door lock manager& Power lock
Door lock	Auto lock
Door relock	Auto lock
F_automatic	C_automatic
F_manual	C_manual
Shift out of Park	Gear in Park
F_speed	C_speed

Table VII
REQUIRES RELATION IN ECPL

Implements:: $implements_{\Psi}(A,\ f)$ if $\exists C_1 \in prov(f), C_2 \in req(f) \cdot C_2 \subseteq C_1 \subseteq A$. The set of features implemented by an architecture A is defined as $Provided\_by_{\Psi}(A) = \{f | implements_{\Psi}(A,\ f)\}$.

Examples : In ECPL, check if $implements_{\Psi}(A_4,$ *Power Lock*$)$ holds.

Component/Interconnection	Feature
Door lock manager & Power lock	Power lock
C_automatic	F_automatic
C_manual	F_manual
Auto lock	Door lock & Door relock
Gear in park	Shift out of park
C_speed	F_speed

Table VIII
PROVIDES RELATION IN ECPL

Architectures / Features	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9
F_1									
F_2				1	1	1	1	1	1
F_3		1	1			1	1	1	1
F_4		1	1			1	1	1	1
F_5				1		1		1	
F_6					1		1		1
F_7		1				1	1		
F_8			1					1	1

Table IX
FEATURE IMPLEMENTATION IN GIVEN SPL.

Architectures / Specifications	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9
S_1				1					
S_2					1				
S_3									
S_4									
S_5									
S_6						1			
S_7							1		
S_8								1	

Table X
SPECIFICATIONS AND THE REALIZING ARCHITECTURES.

Architectures / Specifications	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9
S_1				1		1		1	
S_2					1		1		
S_3						1			
S_4							1		
S_5								1	
S_6						1			
S_7							1		
S_8								1	

Table XI
SPECIFICATIONS AND THEIR COVERING ARCHITECTURES.

Solution : Let P1 = $prov(Power\ Lock)$. From Table II, P1 = $prov(Power\ Lock) = \{\{Door\ Lock\ Manager,\ Power\ Lock\}\}$. Let R1 = $req(Power\ Lock)$. From Table I, R1 = $req(Power\ Lock) = \{\{Door\ Lock\ Manager,\ Power\ Lock\}\}$. Since $R_1 \subseteq P_1 \subseteq A_4$, $implements_{\Psi}(A_4,\ Power\ Lock)$ holds. On other hand $R_1 \subseteq P_1 \nsubseteq A_1$, hence$implements_{\Psi}(A_1,\ Power\ Lock)$ does not hold.

For each feature, we can find the architectures which implement it. The results are listed in Table IX: the 1's in the column corresponding to an architecture gives us the features implemented.

Realization:: Given $A \in \overline{\mathcal{C}}$ and $S \in \overline{\mathcal{F}}$, $Realizes(A,\ S)$ if $S = Provided\_by(A)$.

Example : In ECPL, check if $Realizes(A_4,\ S_1)$ holds.

Solution : The specification S_1 has the features $\{Power\ Lock,\ F\_automatic\}$. From Table IX, $Provided\_by(A_4) = \{Power\ Lock,\ F\_automatic\}$. Since $Provided\_by(A_4) = S_1$, $Realizes(A_4,\ S_1)$ holds. On the other hand,
$Provided\_by(A_5) = \{Power\ Lock,\ F\_manual\} \neq S_1$, hence $Realizes(A_5,\ S_1)$ does not hold.

The Table X shows all the specifications and it's corresponding realized architectures.

Covers:: Given $A \in \overline{\mathcal{C}}$ and $S \in \overline{\mathcal{F}}$, A covers S if $Provided\_by(A) \in \overline{\mathcal{F}} \wedge S \subseteq Provided\_by(A)$.

Example : In ECPL, check $Covers(A_6,\ S_1)$ Hold?

Solution : The specification S_1 has $\{Power\ Lock,\ F\_automatic\}$ features. From Table IX, $Provided\_by(A_6) = \{Power\ Lock,\ Door\ Lock,\ Door\ Relock,\ F\_automatic\}$. Since $Provided\_by(A_6) \in \overline{\mathcal{F}}$ and $S_1 \subseteq Provided\_by(A_6)$, hence $Covers(A_6,\ S_1)$ hold. On the other hand, $Provided\_by(A_5) = \{Power\ Lock,\ F\_manual\} \in \overline{\mathcal{F}}$ but $S_1 \nsubseteq Provided\_by(A_5)$, hence $Covers(A_5,\ S_1)$ does not hold.

Similarly, for all other specifications we can find the architectures which cover the specifications. The Table XI has all the specifications and their covering architectures.

A. SPL Level Properties of ECPL

Completeness:: In ECPL, from Table XI one can observe that every specification in scope $\overline{\mathcal{F}}$ is covered by some architecture in platform $\overline{\mathcal{C}}$. Hence, ECPL is complete.

Soundness:: From Table XI one can observe that the architectures S_1, S_2 and S_3 do not cover any specification in scope $\overline{\mathcal{F}}$. Hence, ECPL is not sound.

Existentially Explicit:: It is observed from Table X that the architectures S_1, S_2, S_6, S_7 and S_8 are realized by the architectures A_4, A_5, A_6, A_7 and A_8 respectively. Hence these specifications are existentially explicit. From the same table, one can observe that the specifications S_3, S_4 and S_5 are not realized by any architecture in the given platform.

Universally Explicit:: In ECPL, from Table X and XI, it is observed that the specifications S_6, S_7 and S_8 are realized by the architectures A_6, A_7 and A_8 respectively, and these are the only architectures which cover the respective specifications. Hence, these specifications are universally explicit. As we have already seen from Table X, the architectures S_3, S_4 and S_5 are not realized at all. The remaining architectures S_1 and S_2 are realized by A_4 and A_5 respectively, but S_1 is also strictly covered (covered but not realized) by architectures A_6 and A_7 and S_2 is strictly covered by A_7. Hence, the specifications S_1, S_2, S_3, S_4 and S_5 are not universally explicit.

Unique Implementation:: In an SPL, a given specification is said to be uniquely implemented if it is covered by exactly one architecture. In ECPL, from Table XI it is found that the specifications S_3, S_4, S_5, S_6, S_7 and S_8 are covered by exactly one architecture (A_6, A_7, A_8, A_6, A_7, A_8 respectively). Hence, these specifications have unique implementation. On the other hand, the specifications S_1 and S_2 have multiple implementations.

Products:: In ECPL, from Table XI we get $Prod(\Psi)$ $=\{\langle S_1, A_4\rangle$, $\langle S_1, A_6\rangle$, $\langle S_1, A_8\rangle$, $\langle S_2, A_5\rangle$, $\langle S_2, A_7\rangle$, $\langle S_3, A_6\rangle$, $\langle S_4, A_7\rangle$, $\langle S_5, A_8\rangle$, $\langle S_6, A_6\rangle$, $\langle S_7, A_7\rangle$, $\langle S_8, A_8\rangle\}$.

Common, live and dead elements:: From the set of products and referring to the tables V and VI, we find that the common elements of ECPL are $\{Power\ Lock^1$, $Door\ Lock\ Manager$, $Unlock\ Driver\ Door$, $Unlock\ all\ doors$, $Lock\ all\ doors$, $Power\ Lock^2$, $Courtesy\ switch$, $Key\ signal$, $Sill\ door\ signal\}$. $Power\ Lock^1$ is the feature and $Power\ Lock^2$ is the component.

The live elements for $Prod(\Psi)$ are $\{Power\ Lock^1$, $Door\ Lock$, $Door\ Relock$, $F\_automatic$, F_manual, F_speed, $Shift\ out\ of\ Park$, $Door\ Lock\ Manager$, $Unlock\ Driver\ Door$, $Unlock\ all\ doors$, $Lock\ all\ doors$, $Auto\ Lock$, $Power\ Lock^2$, $Courtesy\ switch$, $Key\ signal$, $Sill\ door\ signal$, $C\_automatic$, C_manual, $Gear\ in\ park$, $C\_speed\}$. The only dead element is $Manual\ Lock$.

Superfluous Component:: There are no superfluous components in ECPL. For example, consider the element $AutoLock$. The specification S_1 is covered by architectures A_4, A_6 and A_8. If architectures A_6 and A_8, which include $AutoLock$, are removed, then S_1 is still in the product (being implemented by A_4). However, A_6 is the only architecture covering S_3. Hence, when A_6 is removed, $\langle S_3, A_6\rangle$ is removed from the list of products. This implies that $AutoLock$ is not superfluous.

Redundant Component:: A component is redundant if it is not contributing to any feature in any architecture in the platform. In ECPL, there are not any redundant component. Let us assume we have a component called $Door\ Relock_{Alt}$ such that $\{Door\ Relock_{Alt}$, $Auto\ Lock$‘$\}$ provides the feature $Door\ Relock$. This component would be redundant because $Auto\,lock$ already provide the feature $Door\ Relock$.

Critical Component:: In ECPL, all the components are critical. Let us remove the component $C\_automatic$ from architecture A_4. Then, $implements_{\Psi}(A_4, F\_automatic))$ will not hold. Hence, we can say that the component $C\_automatic$ is critical for feature $F\_automatic$.

Emerging Features:: In ECPL, the specification S_4 is not realized by any architecture but it is covered by A_7. So the set of emerging features is $Provided\_by(A_7) - S_4$=$\{Door\ relock\}$.

Properties and Formulae	Test 1	Test 2	Test 3	Average Time(ms)
$Implements$	3	2	2	2.33
$realizes$	2	2	2	2
$covers$	3	2	2	2.33
$complete$	3	2	2	2.33
$sound$	4	3	3	3.33
$existentially\ explicit$	3	2	3	2.67
$critical$	3	3	3	3
$extended\ features$	2	2	2	2

Table XII
TIME COMPLEXITY FOR PROPERTIES AND FORMULAE

B. Performance

We have recorded the time required to check the satisfiability of the formulae for some analysis problems using QuBE (Refer Table XII). Each formula has been run three times and the average time is calculated. The performance of QuBE seems quite good for small SPLs the size of ECPL.

V. ANALYSIS BETWEEN THE SPECIFICATION AND THE IMPLEMENTATION PERSPECTIVES

In the literature, different analysis problems in SPL are usually encoded as propositional satisfiability problems[14] and SAT solvers such as Yices, Bddsolve[15] etc. are used to solve the problems. However, looking at the definition of $implements$ and the subsequent problems, we observe that there is quantification over the features and components which can be encoded as propositions. In fact, we show in the following that it is possible to transform the analysis problems of the previous section into QBF formula such that the questions have an affirmative answer iff the corresponding QBF formulae hold.

1) Let $\mathcal{C} = \{c_1, \ldots, c_n\}$ be the set of all components and let $\mathcal{F} = \{f_1, \ldots, f_m\}$ be the set of all features. A subset of $\mathcal{F}$ is a specification, while a subset of $\mathcal{C}$ is called an architecture. A platform is a set of architectures $\overline{\mathcal{C}} \subseteq \mathcal{P}(\mathcal{C})$. A scope is a set of specifications $\overline{\mathcal{F}} \subseteq \mathcal{P}(\mathcal{F})$.
2) Given an architecture $C = \{c_1, \ldots, c_k\}$, let $Prop(C)$ be the tuple of propositions
$$Prop(C)(i) = \begin{cases} c_i \text{ if } c_i \in C \\ \neg c_i \text{ if } c_i \notin C \end{cases}$$
Thus, $Prop(C)$ is an n-tuple made up of 0's and 1's. The tuple $Prop(F)$ for a specification F can be defined similarly.
3) Let f be a feature. Let $prov(f) = \{S_1, S_2 \ldots, S_k\}$. Each S_j is a set of components that provides f. Then we define $formula\_prov(f)$ as $\bigvee_j \bigwedge_{c_i \in S_j} c_i$. $formula\_prov(f)$ is satisfiable whenever there is some set S_j of components that provide feature f. If the set $prov(f)$ is undefined(empty), then $formula\_prov(f)$ is FALSE, since there are no components that provide feature f.

4) Let f be a feature. Let $req(f) = \{S_1, S_2 \ldots, S_k\}$. f requires at least one set S_j of components for its implementation. Then, we define $formula\_req(f) = \bigvee_j \bigwedge_{c_i \in S_j} c_i$. $formula\_req(f)$ is satisfiable iff $req(f)$ has at least one set (say S_j) of its required components. If $req(f)$ is empty or undefined, then $formula\_req(f)$ is TRUE, since there are no requirements for f.
5) Let f be a feature and let $prov(f) = \{S_1, S_2 \ldots, S_k\}$. Given a tuple of component parameters $(c'_1, \ldots, c'_n)$ where each c'_i is 0 or 1, and a feature f, we define the formula $f\_implements(c'_1, \ldots, c'_n, f)$ as

$$\forall c_1 \ldots c_n\{[\bigwedge_{i=1}^{n}(c'_i \Rightarrow c_i)] \Rightarrow formula\_prov(f)\}$$

Whenever the truth values of c_i agree with those of the variables of some S_j in $prov(f)$, or correspond to a superset of some S_j in $prov(f)$, the formula $formula\_prov(f)$ will hold good.
6) Let $F = \{f_1, f_2, \ldots, f_l\}$ be a specification. For each f_i, let $prov(f_i) = \{S_{i1}, \ldots, S_{ik}\}$ be defined. Consider a tuple of component parameters $(c'_1, \ldots, c'_n)$ and a tuple of feature parameters $(f'_1, \ldots, f'_m)$. Here again, each c'_i, f'_j is a zero or a 1. Define $f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$ as

$$\bigwedge_{i=1}^{m}(f'_i \Rightarrow f\_implements(c'_1, \ldots, c'_n, f_i))$$

Define $f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$ as

$$\bigwedge_{i=1}^{m}(f'_i \Leftrightarrow f\_implements(c'_1, \ldots, c'_n, f_i))$$

7) Let $\Psi = (\overline{\mathcal{F}}, \overline{\mathcal{C}}, \mathcal{T})$ be an SPL. Let $\overline{\mathcal{C}} = \{S_1, \ldots, S_k\}$. Given a tuple of component parameters $c'_1, \ldots, c'_n$ where each c'_i is 0 or 1, the predicate $C_I(c'_1, \ldots, c'_n)$ is defined as

$$\bigvee_j \bigwedge_{c_i \in Prop(S_j)} c'_i$$

Then $C_I(c'_1, \ldots, c'_n)$ is satisfied iff $\{c'_k \mid c'_k = 1\} = S_l$ for some $S_l \in \overline{\mathcal{C}}$. $C_F(f'_1, \ldots, f'_m)$ is defined similarly.

Lemma 1. *(Internal Consistency of Traceability) Consider a canonical SPL. Let TCF, the trace consistency formula be defined as* $\forall c_1 \ldots c_n. \bigwedge_{f \in F}[f\_prov(f) \Rightarrow f\_req(f)]$. *Then, $\mathcal{T}$ is internally consistent iff TCF is true.*

Lemma 2. *(Implements) Given a canonical SPL, a set of components C, and a feature f, $implements(C, f)$ iff $f\_implements(c'_1, \ldots, c'_n, f)$ where $Prop(C) = (c'_1, \ldots, c'_n)$.*

Lemma 3. *(Realizes, Covers) Given a set of components C and a set of features F, let $Prop(C) = (c'_1, \ldots, c'_n)$ and $Prop(F) = (f'_1, \ldots, f'_m)$. Then the following statements hold:*

1) *C covers F iff $f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$*
2) *C realizes F iff $f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$*

Lemma 4. *(Completeness, Soundness) Given an SPL, the SPL is complete iff*
$\forall f'_1 \ldots f'_m[C_F(f'_1, \ldots, f'_m) \Rightarrow \exists c'_1 \ldots c'_n[C_I(c'_1, \ldots, c'_n) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)]$
Given an SPL, the SPL is sound iff
$\forall c_1 \ldots c_n[C_I(c_1, \ldots, c_k)] \Rightarrow \exists f_1 \ldots f_j[C_F(f_1, \ldots, f_j) \wedge f\_covers(c_1, \ldots, c_k, f_1, \ldots, f_j)]$

Lemma 5. *(Existentially Explicit Features) Given a set of features F, let $Prop(F) = (f'_1, \ldots, f'_m)$. Then F is existentially explicit iff* $\exists c'_1 \ldots c'_n[C_I(c'_1, \ldots, c'_n) \wedge f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)]$.

Lemma 6. *(Universally Explicit Features) Given a set of features F, let $Prop(F) = (f'_1, \ldots, f'_m)$. Then F is universally explicit iff* $\exists c'_1 \ldots c'_n[C_I(c'_1, \ldots, c'_n) \wedge f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \wedge \forall c'_1 \ldots c'_n\{[(C_I(c'_1, \ldots, c'_n) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \Rightarrow f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)\}$.

Lemma 7. *(Unique Implementation) Given a set of features F, let $Prop(F) = (f'_1, \ldots, f'_m)$. Then F has a unique implementation iff* $\exists c'_1 \ldots c'_n[C_I(c'_1, \ldots, c'_n) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \wedge \forall d'_1 \ldots d'_n\{[C_I(d'_1, \ldots, d'_n) \wedge f\_covers(d'_1, \ldots, d'_n, f'_1, \ldots, f'_m)] \Rightarrow (\wedge_{l=1}^{n}(d'_i \Leftrightarrow c'_i)\}$

Lemma 8. *(Common, live and dead elements)*

1) *A component c is common iff*
$\forall c'_1, \ldots, c'_n, f'_1, \ldots, f'_m\{[C_I(c'_1, \ldots, c'_n) \wedge C_F(f'_1, \ldots, f'_m) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \Rightarrow c\}$ *holds.*
2) *A component c is live iff*
$\exists c'_1, \ldots, c'_n, f'_1, \ldots, f'_m\{[C_I(c'_1, \ldots, c'_n) \wedge C_F(f'_1, \ldots, f'_m) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m) \wedge c\}$
3) *A component c is dead iff*
$\forall c'_1, \ldots, c'_n, f'_1, \ldots, f'_m\{[C_I(c'_1, \ldots, c'_n) \wedge C_F(f'_1, \ldots, f'_m) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \Rightarrow \neg c\}$ *holds.*

Lemma 9. *(Superflous) A component c_i is superflous iff* $\forall c'_1, \ldots, c'_n, f'_1, \ldots, f'_m\{[c'_i \wedge C_I(c'_1, \ldots, c'_n) \wedge C_F(f'_1, \ldots, f'_m) \wedge f\_covers(c'_1, \ldots, c'_i, \ldots, c'_n, f'_1, \ldots, f'_m)] \Rightarrow \exists d'_1, \ldots, d'_n[\neg d'_i \wedge C_I(d'_1, \ldots, d'_n) \wedge f\_covers(d'_1, \ldots, d'_n, f'_1, \ldots, f'_m)]\}$.

Lemma 10. *(Redundant) A component c_i is redundant iff* $\forall c'_1, \ldots, c'_n f'_1 \ldots, f'_m\{[c'_i \wedge C_I(c'_1, \ldots, c'_n) \wedge C_F(f'_1, \ldots, f'_m) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \Rightarrow$

$\exists d'_1 \ldots d'_n[\neg d'_i \wedge (\bigwedge_{i=1}^{n} c'_i \Rightarrow \bigwedge d'_i) \wedge C_I(d'_1, \ldots, d'_n) \wedge f\_covers(d'_1, \ldots, d'_n, f'_1, \ldots, f'_m)]\}$.

Lemma 11. *(Critical) A component c is critical for f_j iff $\forall c'_1, \ldots, c'_n\{[C_I(c'_1, \ldots, c'_n) \wedge f\_implements(c'_1, \ldots, c'_n, f_j)] \Rightarrow c\}$.*

Lemma 12. *(Extends) Let F and F' be subsets of features. Let $Prop(F) = (f_1, \ldots, f_m)$ and $Prop(F') = (f'_1, \ldots, f'_m)$. Then F' extends F iff $\bigwedge_{i=1}^{m}(f_i \Rightarrow f'_i)$ is true. F' is extendable iff $\exists f'_1, \ldots, f'_m[\bigwedge_{i=1}^{m} f_i \Rightarrow f'_i)]$.*

Theorem 15. *Given an SPL Ψ, each of the properties listed in Table XIII holds good iff the corresponding formulae evaluate to true.*

Proof: The detailed proof is given in the full version of the paper. ■

VI. Implementation

In this section, we give some details of the implementation of the theory developed, using off-the-shelf QSAT solvers. We also illustrate the encoding of the analysis problems in QBF and their solutions through a small example.

A. QBF and QDIMACS format

Quantified Boolean Formulae (QBF) are generalized form of propositional formulae with quantification (existential and universal) over the propositional symbols. The boolean satisfiability problem for propositional formulae is then naturally extended to QBF satisfiability problem (QSAT).

Most QBF solvers follow QDimacs, a standard input and output file format. QDimacs Format is built on top of the DIMACS standard for SAT Solver. A QDimacs file representing a QBF has three parts: Preamble, Prefix and Matrix. The notations use a unique indexing of all the propositional variables occurring in the QBF.

1) $Preamble$: The Preamble contains different types of information about the file, namely,
   a) $Comments$: Each comment line should start with lower case character 'c'. There can be multiple comment lines in the File.
      Format:
      `c COMMENT_STRING`
      Example:
      `c Testing QBF formulae.`
      `c qdimac file for completeness.`
   b) $Problem\ Line$: There is only one problem line in each QDimacs File. The problem line starts with the lower case character 'p' followed by the string 'cnf', which denotes that the given formula is in conjunctive normal form (CNF). The 'cnf' string is followed by variables count and clauses count.
      Format:
      `p cnf VAR_COUNT CLA_COUNT`
      Example:
      `p cnf 4 2`
2) $Prefix$: The Prefix lines are used to represent the quantifiers in the Formula. Each Prefix line starts with a lower case character 'a' or 'e'; 'a' represents universal quantifier and 'e' represents existential quantifier. Quantifiers are followed by the indices of variables. Each prefix line ends with '0'.
   Example:
   `a 1 2 0`
   `e 3 4 0`
3) $Matrix$: Each line in matrix represents a clause and should end with '0'. Each propositional variable in clause is represented by it's corresponding unique index. The complement of a variable is represented by negation of the index.
   Example:
   1 3 0
   2 -4 0

As an example, the QDimacs format for the formula $\forall X \exists Y((X \vee \neg Y) \wedge (\neg X \vee Y)$ is as follows. The first line is a comment line. The second one is the problem line which mentions that there are two variables and two clauses. The third line represents the universal quantification of X and the fourth line represents the existential quantification of Y. The fifth line represents the first clause $(X \vee \neg Y)$ and the sixth line represents the second clause $(\neg X \vee Y)$.

```
c Illustration
p cnf 2 2
a 1 0
e 2 0
1 -2 0
-1 2 0
```

QuBE is a solver for Quantified Boolean Formulas (QBFs). It accepts QBFs in QDimacs format and returns TRUE if the formula is satisfiable, and FALSE otherwise. We have developed a tool called CNF2QDIMAC converter. The tool converts QBFs in CNF to QDimacs format which can be given as input to QuBE. Conversion of arbitrary QBFs to CNF is done using some online tools.

B. An Illustrative Example

Consider the following SPL $\Psi = (\overline{\mathcal{C}}, \overline{\mathcal{F}}, \mathcal{T})$ with $\overline{\mathcal{C}} = \{\{c_1, c_2\}, \{c_3, c_4\}\}$ and $\overline{\mathcal{F}} = \{\{f_1, f_2\}, \{f_3\}\}$. Thus, there are 4 components and 3 features. Further, let the traceability relation $\mathcal{T}$ be given as follows:

- $prov(f_1) = \{\{c_1, c_2\}, \{c_3\}\}, req(f_1) = \{\{\{c_1\}, \{c_3\}\}$
- $prov(f_2) = \{\{c_2\}\}, req(f_1) = \{\{c_2\}\}$
- $prov(f_3) = \{\{c_1, c_4\}\}, req(f_3) = \{\{c_4\}\}$

Let us answer the following questions using the logic formulation with the help of the QuBE tool.

Properties	Formula
$Implements(C, f)$ $Prop(C) = (c'_1, \ldots, c'_n)$	$f\_implements(c'_1, \ldots, c'_n, f)$
C covers F, $Prop(C) = (c'_1, \ldots, c'_n)$ C realizes F, $Prop(F) = (f'_1, \ldots, f'_m)$	$f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$ $f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$
Ψ complete	$\forall f'_1 \ldots f'_m\{C_F(f'_1, \ldots, f'_m) \Rightarrow \exists c'_1 \ldots c'_n[C_I(c'_1, \ldots, c'_n) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)]\}$
Ψ sound	$\forall c'_1 \ldots c'_n\{C_I(c'_1, \ldots, c'_n)] \Rightarrow \exists f'_1 \ldots f'_m[C_F(f'_1, \ldots, f'_m) \wedge f\_covers(c'_1, \ldots, c'_k, f_1, \ldots, f_j)]\}$
F existentially explicit $Prop(F) = (f'_1, \ldots, f'_m)$	$\exists c'_1 \ldots c'_n[C_I(c'_1, \ldots, c'_n) \wedge f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)]$
F universally explicit $Prop(F) = (f'_1, \ldots, f'_m)$	$\exists c'_1 \ldots c'_n[C_I(c'_1, \ldots, c'_n) \wedge f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \wedge \forall c'_1 \ldots c'_n\{[(C_I(c'_1, \ldots, c'_n) \wedge$ $f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \Rightarrow f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)\}$.
F has unique implementation $Prop(F) = (f'_1, \ldots, f'_m)$	$\exists c'_1 \ldots c'_n[C_I(c'_1, \ldots, c'_n) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \wedge$ $\forall d'_1 \ldots d'_n\{[C_I(d'_1, \ldots, d'_n) \wedge f\_covers(d'_1, \ldots, d'_n, f'_1, \ldots, f'_m)] \Rightarrow (\wedge_{l=1}^{n}(d'_i \Leftrightarrow c'_i)\}$
c common	$\forall c'_1, \ldots, c'_n, f'_1, \ldots, f'_m\{[C_I(c'_1, \ldots, c'_n) \wedge C_F(f'_1, \ldots, f'_m) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \Rightarrow c\}$
c live	$\exists c'_1, \ldots, c'_n, f'_1, \ldots, f'_m\{[C_I(c'_1, \ldots, c'_n) \wedge C_F(f'_1, \ldots, f'_m) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m) \wedge c\}$
c dead	$\forall c'_1, \ldots, c'_n, f'_1, \ldots, f'_m\{[C_I(c'_1, \ldots, c'_n) \wedge C_F(f'_1, \ldots, f'_m) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \Rightarrow \neg c\}$
c_i superfluous	$\forall c'_1, \ldots, c'_n, f'_1, \ldots, f'_m\{[c'_i \wedge C_I(c'_1, \ldots, c'_n) \wedge C_F(f'_1, \ldots, f'_m) \wedge f\_covers(c'_1, \ldots, c'_i, \ldots, c'_n, f'_1, \ldots, f'_m)] \Rightarrow$ $\exists d'_1, \ldots, d'_n[\neg d'_i \wedge C_I(d'_1, \ldots, d'_n) \wedge f\_covers(d'_1, \ldots, d'_n, f'_1, \ldots, f'_m)]\}$
c_i redundant	$\forall c'_1, \ldots, c'_n f'_1 \ldots, f'_m\{[c'_i \wedge C_I(c'_1, \ldots, c'_n) \wedge C_F(f'_1, \ldots, f'_m) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \Rightarrow$ $\exists d'_1 \ldots d'_n[\neg d'_i \wedge (\bigwedge_{i=1}^{n} c'_i \Rightarrow \bigwedge d'_i) \wedge C_I(d'_1, \ldots, d'_n) \wedge f\_covers(d'_1, \ldots, d'_n, f'_1, \ldots, f'_m)]\}$
c critical for f_j	$\forall c'_1, \ldots, c'_n\{[C_I(c'_1, \ldots, c'_n) \wedge f\_implements(c'_1, \ldots, c'_n, f_j)] \Rightarrow c\}$

Table XIII
PROPERTIES AND FORMULAE

1) Does $C = \{c_1, c_2\}$ implement f_1? Clearly, the answer is YES. In the logic formalism, $f\_implements(1, 1, 0, 0, f_1)$ is defined as $\forall c_1 c_2 c_3 c_4\{[(1 \Rightarrow c_1) \wedge (1 \Rightarrow c_2) \wedge (0 \Rightarrow c_3) \wedge (0 \Rightarrow c_4)] \Rightarrow f\_prov(f_1)\}$ where $f\_prov(f_1)$ = $(c_1 \wedge c_2) \vee c_3$. The formula when simplified is $\forall c_1 c_2 c_3 c_4((c_1 \wedge c_2) \Rightarrow ((c_1 \wedge c_2) \vee c_3)$. It is easy to see that the formula evaluates to true. Hence QuBE returns an affirmative answer.
   Now consider $C = \{c_3\}$. Does C implement f_3? Clearly, the answer is NO. In the logic formalism, $f\_implements(0, 0, 1, 0, f_3)$ is defined as $\forall c_1 c_2 c_3 c_4\{[(0 \Rightarrow c_1) \wedge (0 \Rightarrow c_2) \wedge (1 \Rightarrow c_3) \wedge (0 \Rightarrow c_4)] \Rightarrow f\_prov(f_3)\}$ where $f\_prov(f_3)$ = $(c_1 \wedge c_4)$. The simplified formula is $\forall c_1 c_2 c_3 c_4(c_3 \Rightarrow (c_1 \wedge c_4))$. The assignment $c_3 = 1$, $c_1 = 0$ evaluates the quantifier-free formula to false. Hence QuBE returns a negative answer.
2) Consider $C = \{c_1, c_2\}$. Does C realize $\{f_1, f_2\}$? Clearly, the answer is YES. In the logic formalism, $f\_realizes(1, 1, 0, 0, 1, 1, 0)$ is defined as $([1 \Leftrightarrow f\_implements(1, 1, 0, 0, f_1)] \wedge [1 \Leftrightarrow f\_implements(1, 1, 0, 0, f_2)] \wedge [0 \Leftrightarrow f\_implements(1, 1, 0, 0, f_3)] \wedge [0 \Leftrightarrow f\_implements(1, 1, 0, 0, f_4)]$.

   Now, $f\_implements(1, 1, 0, 0, f_1)$ is defined as $\forall c_1 c_2 c_3 c_4\{[(1 \Rightarrow c_1) \wedge (1 \Rightarrow c_2) \wedge (0 \Rightarrow c_3) \wedge (0 \Rightarrow c_4)] \Rightarrow f\_prov(f_1)\}$ where $f\_prov(f_1)$ is defined as $(c_1 \wedge c_2) \vee c_3$. Clearly, $f\_implements(1, 1, 0, 0, f_1)$ holds. Thus, we have $[1 \Leftrightarrow f\_implements(1, 1, 0, 0, f_1)]$ is true. Similarly, it can be seen that $[1 \Leftrightarrow f\_implements(1, 1, 0, 0, f_2)]$ is true.

   Likewise, $f\_implements(1, 1, 0, 0, f_3)$ is $\forall c_1 c_2 c_3 c_4[(1 \Rightarrow c_1) \wedge (1 \Rightarrow c_2) \wedge (0 \Rightarrow c_3) \wedge (0 \Rightarrow c_4)] \Rightarrow (c_1 \wedge c_4)$, which is false. Hence, $[0 \Leftrightarrow f\_implements(1, 1, 0, 0, f_3)]$ is true.

   Similarly, $f\_implements(1, 1, 0, 0, f_4)$ is $\forall c_1 c_2 c_3 c_4\{[(1 \Rightarrow c_1) \wedge (1 \Rightarrow c_2) \wedge (0 \Rightarrow c_3) \wedge (0 \Rightarrow c_4)] \Rightarrow (c_4)\}$, which is false. Hence, $[0 \Leftrightarrow f\_implements(1, 1, 0, 0, f_4)]$ is true. Thus, we have the answer true from QuBE.
   Now consider the question: does C realize f_1? Clearly, C covers f_1, but realizes $\{f_1, f_2\}$. Again, the logic formalism for the same is $f\_realizes(1, 1, 0, 0, 1, 0, 0)$, which is defined as $([1 \Leftrightarrow f\_implements(1, 1, 0, 0, f_1)] \wedge [0 \Leftrightarrow f\_implements(1, 1, 0, 0, f_2)] \wedge [0 \Leftrightarrow f\_implements(1, 1, 0, 0, f_3)] \wedge [0 \Leftrightarrow f\_implements(1, 1, 0, 0, f_4)]$.
   As seen above, clearly, $[1 \Leftrightarrow f\_implements(1, 1, 0, 0, f_1)]$ holds. However, we have $f\_implements(1, 1, 0, 0, f_2)$ is true since $prov(f_2) = \{\{c_2\}\}$. Then, we do not have $[0 \Leftrightarrow f\_implements(1, 1, 0, 0, f_2)]$. Therefore, QuBE returns false.
3) Is the given SPL complete? That is, for every $F \in \overline{\mathcal{F}}$, does there exist some $C \in \overline{\mathcal{C}}$ such that $Covers(C, F)$? Clearly, the answer is NO since there is no $C \in \overline{\mathcal{C}}$ covering $\{f_3\} \in \overline{\mathcal{F}}$. The formula for this is $\forall f'_1 f'_2 f'_3[C_F(f'_1, f'_2, f'_3) \Rightarrow \exists c'_1 c'_2 c'_3 c'_4[C_I(c'_1, \ldots, c'_4) \wedge f\_covers(c'_1, \ldots, c'_4, f'_1, \ldots, f'_3)]$. This expands

out to
$C_F(1,1,1) \Rightarrow \exists c'_1,c'_2c'_3c'_4[C_I(c'_1,\ldots,c'_4) \wedge f\_covers(c'_1,\ldots,c'_4,1,1,1)]$ and
$C_F(1,1,0) \Rightarrow \exists c'_1,c'_2c'_3c'_4[C_I(c'_1,\ldots,c'_4) \wedge f\_covers(c'_1,\ldots,c'_4,1,1,0)]$ and
$C_F(1,0,1) \Rightarrow \exists c'_1,c'_2c'_3c'_4[C_I(c'_1,\ldots,c'_4) \wedge f\_covers(c'_1,\ldots,c'_4,1,0,1)]$ and
$C_F(0,1,1) \Rightarrow \exists c'_1,c'_2c'_3c'_4[C_I(c'_1,\ldots,c'_4) \wedge f\_covers(c'_1,\ldots,c'_4,0,1,1)]$ and
$C_F(0,0,1) \Rightarrow \exists c'_1,c'_2c'_3c'_4[C_I(c'_1,\ldots,c'_4) \wedge f\_covers(c'_1,\ldots,c'_4,0,0,1)]$ and
$C_F(0,1,0) \Rightarrow \exists c'_1,c'_2c'_3c'_4[C_I(c'_1,\ldots,c'_4) \wedge f\_covers(c'_1,\ldots,c'_4,0,1,0)]$ and
$C_F(1,0,0) \Rightarrow \exists c'_1,c'_2c'_3c'_4[C_I(c'_1,\ldots,c'_4) \wedge f\_covers(c'_1,\ldots,c'_4,1,0,0)]$ and
$C_F(0,0,0) \Rightarrow \exists c'_1,c'_2c'_3c'_4[C_I(c'_1,\ldots,c'_4) \wedge f\_covers(c'_1,\ldots,c'_4,0,0,0)]$.

Among these, $C_F(1,1,0)$, $C_F(0,0,1)$ evaluates to true. The rest evaluate to false - hence the formula involving them holds.

Now, consider $C_F(1,1,0)$. Then we must check whether $\exists c'_1c'_2c'_3c'_4[C_I(c'_1,\ldots,c'_4) \wedge f\_covers(c'_1,\ldots,c'_4,1,1,0)]$ holds. The tuple $(1,1,0,0)$ as well as $(0,0,1,1)$ satisfy $C_I(c'_1,c'_2,c'_3,c'_4)$. Hence, these are the only two tuples that we need to examine for (c'_1,c'_2,c'_3,c'_4). Consider $(1,1,0,0)$. Then $[C_I(1,1,0,0) \wedge f\_covers(1,1,0,0,1,1,0)]$ evaluates to $true \wedge [1 \Rightarrow f\_implements(1,1,0,0,f_1)] \wedge [1 \Rightarrow f\_implements(1,1,0,0,f_2)] \wedge$
$[0 \Rightarrow f\_implements(1,1,0,0,f_3)]$. Clearly, this is true, as $\{c_1,c_2\}$ covers $\{f_1,f_2\}$.

Now consider $C_F(0,0,1)$. Then we must check $\exists c'_1c'_2c'_3c'_4[C_I(c'_1,\ldots,c'_4) \wedge f\_covers(c'_1,\ldots,c'_4,0,0,1)]$ holds. Again, consider the two possibilities for $C_I(c'_1,c'_2,c'_3,c'_4)$. Look at $C_I(1,1,0,0)$ first. Then we have to check if $f\_covers(1,1,0,0,0,0,1)$ is true. This is $[0 \Rightarrow f\_implements(1,1,0,0,f_1)] \wedge [0 \Rightarrow f\_implements(1,1,0,0,f_2)] \wedge [1 \Rightarrow f\_implements(1,1,0,0,f_3)]$. Clearly, $f\_implements(1,1,0,0,f_3)$ does not hold since $prov(f_3) = \{c_1,c_4\}$ and c_4 can be assigned 0 in this formula. Now consider the second assignment $(0,0,1,1)$. Then again, $C_I(0,0,1,1)$ holds. Now check if $f\_covers(0,0,1,1,0,0,1)$ holds. That is, $[0 \Rightarrow f\_implements(0,0,1,1,f_1)] \wedge [0 \Rightarrow f\_implements(0,0,1,1,f_2)] \wedge [1 \Rightarrow f\_implements(0,0,1,1,f_3)]$. Since $prov(f_3) = \{\{c_1,c_4\}\}$, $f\_implements(0,0,1,1,f_3)$ is false. Thus, this does not hold good as well.

Therefore, for $\{f_3\}$ (equivalently, $C_F(0,0,1)$), there is no $C_I(c'_1,c'_2,c'_3,c'_4)$ which realizes $\{f_3\}$. Hence, QuBE returns false. Then, we can conclude that the SPL is not complete.

VII. Results of Analyses on The ECPL Case-study

In this section, we analyze some properties of the ECPL example using QUBE. The platform $\overline{\mathcal{C}}$ contains the following architectures:

1) C_1 = { Door Lock Manager, Unlock Driver Door, Unlock all doors, Lock all doors}
2) C_2 = {Door lock manager, Unlock driver door, Unlock all doors, Lock all doors, AutoLock, Speed}
3) C_3 = { Door lock manager, Unlock driver door, Unlock all doors, Lock all doors, AutoLock, Gear in park }
4) C_4 = { Door lock manager, Unlock driver door, Unlock all doors, Lock all doors, Power Lock, Courtesy switch, Key signal, silldoor signal, Automatic}
5) C_5 = { Door lock manager, Unlock driver door, Unlock all doors, Lock all doors, Power Lock, Courtesy switch, Key signal, silldoor signal, Manual}
6) C_6 = { Door lock manager, Unlock driver door, Unlock all doors, Lock all doors, AutoLock, Speed, Power Lock, Courtesy switch, Key signal, silldoor signal, Automatic}
7) C_7 = { Door lock manager, Unlock driver door, Unlock all doors, Lock all doors, AutoLock, Speed, Power Lock, Courtesy switch, Key signal, silldoor signal, Manual}
8) C_8 = {Door lock manager, Unlock driver door, Unlock all doors, Lock all doors, AutoLock, Gear in park, Power Lock, Courtesy switch, Key signal, silldoor signal, Automatic}
9) C_9 = { Door lock manager, Unlock driver door, Unlock all doors, Lock all doors, AutoLock, Gear in park, Power Lock, Courtesy switch, Key signal, silldoor signal, Manual}

Consider the following specifications in the scope $\overline{\mathcal{F}}$.

1) F_1 = {Power Lock, f_automatic}
2) F_2 = {Power Lock, f_automatic, Door Lock, Shift outof Park, Door relock}.

1) Does C_1 realize F_1? The formula to check is $[1 \Leftrightarrow f\_implements(1,1,1,1,0,\ldots,0$,PowerLock)] $\wedge$ [1 $\Leftrightarrow$ $f\_implements(1,1,1,1,0,\ldots,0,$ f_automatic)] $\wedge \ldots$ [0 $\Leftrightarrow$ $f\_implements(1,1,1,1,0,\ldots,0$,Door relock)]

   Lets look at $f\_implements(1,1,1,1,0,\ldots,0$,PowerLock). Let $c_1 = DoorLockManager$, $c_2 = UnLockDriverDoor$, $c_3 = Unlockalldoors$, $c_4 = Lockalldoors$, $c_5 = PowerLock$. This is defined as $\forall c_1,\ldots,c_n\{([1 \Rightarrow c_1] \wedge [1 \Rightarrow c_2] \wedge [1 \Rightarrow c_3] \wedge [1 \Rightarrow c_4] \wedge [0 \Rightarrow c_5] \ldots [0 \Rightarrow c_n]) \Rightarrow (c_1 \wedge c_5)\}$. Clearly, this does not hold (for $c_5 = 0$, the formula does not hold).

Hence, QUBE returns false.

2) Is ECPL sound? If so, then for every $C_i \in \overline{\mathcal{C}}$, we can find a specification F_i such that $Covers(C_i, F_i)$. The formulae for this is

$$\forall c_1 \dots c_n[C_I(c_1, \dots, c_n)] \Rightarrow \exists f_1 \dots f_m[C_F(f_1, \dots, f_m) \wedge f\_covers(c_1, \dots, c_n, f_1, \dots, f_m)]$$

Consider the tuple $(1, 1, 1, 1, 0, \dots, 0)$ where the first four entries are 1, and the rest are zero. This corresponds to C_1. Clearly, $C_I(1, 1, 1, 1, 0, \dots, 0)$. Lets look at $f\_covers(1, 1, 1, 1, 0, \dots, 0, f'_1, \dots, f'_m)$. It is easy to see that $f\_implements(1, 1, 1, 1, 0, \dots, 0, f)$ does not hold good for any f since $c_1 = DoorLockManager$ does not provide any features alone, and $c_i, i > 0$ do not provide any features. Thus, the formula does not hold good, and QUBE returns false. Hence, the ECPL is not sound.

3) Is F_1 universally explicit? If so, then any $C_i \in \overline{\mathcal{C}}$ which covers F_1 must realize F_1; moreover, there must be atleast one $C \in \overline{\mathcal{C}}$ which covers it. The formula for this is

$$\begin{array}{ll} \exists c'_1 \dots c'_n[C_I(c'_1, \dots, c'_n) & \wedge \\ f\_realizes(c'_1, \dots, c'_n, f'_1, \dots, f'_m)] & \wedge \\ \forall c'_1 \dots c'_n\{[(C_I(c'_1, \dots, c'_n) & \wedge \\ f\_covers(c'_1, \dots, c'_n, f'_1, \dots, f'_m)] & \Rightarrow \\ f\_realizes(c'_1, \dots, c'_n, f'_1, \dots, f'_m)\}. & \end{array}$$

Let us denote c_1=Door lock manager, c_2=AutoLock, c_3=Power Lock, c_4=Gear in Park and c_5=Automatic, c_6=Unlock driver door, c_7=Unlock all doors, c_8=Lock all doors, c_9=Courtsey switch, c_{10}=Key signal, c_{11}=sill door signal, c_{12}=Speed and c_{13}=Manual. Similarly, let f_1=Power Lock and f_2=f_automatic. Consider the component tuple $(1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0)$.

Then we have $C_I(1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0)$.($C_4$ corresponds to this set) and $f\_realizes(1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, \dots, 0)$ (C_4 realizes F_1). Corresponding to this tuple, Consider the component tuple $(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0)$. Clearly, $C_I((1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0)$ (C_8 corresponds to this). As $C_4 \subseteq C_8$, C_8 covers F_1. However, $f\_realizes(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, \dots, 0)$ does not hold since :

Consider $0 \Leftrightarrow$
$f\_implements((1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, ShiftoutofPark)$,
a conjunct in
$f\_realizes(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, \dots, 0)$.
Now, it can be seen that $f\_implements((1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0,$ Shiftout of Park) holds, since the component Gear in Park provides the feature Shift out of Park. Hence, this conjunct does not hold good. Hence, $f\_realizes(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, \dots, 0)$ does not hold.

Hence, QUBE returns false. Thus, for the component tuple $(1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0)$ which realizes F_1, there exists a component tuple which covers, but does not realize F_1. Hence, F_1 is not universally explicit.

VIII. Conclusion

In this report, we have given a new definition for products in a Software Product Line, based on the notion of derivability of feature specifications from component architectures. The traceability relation between features and components plays a central role in this definition. We show that our definition is different from the consistency based definition of SPL products and captures the implementation relation in a more natural way. In the light of this, we define a set of analysis problems for the SPLs. We show that these problems can be formulated as Quantified Boolean Formulae and can be solved using QSAT tools such as QUBE.

We have demonstrated the feasibility of our approach through a small fragment of an industrial SPL. The scalability of the above approach for complete SPLs is yet to be studied. Since QSAT problem is PSPACE-complete, generic QSAT solvers may not scale well. However, one observes that the formulas for the analyses have very specific structure which can be exploited for efficient QSAT solving.

The proposed semantic model of the SPL treats specifications and architectures as sets of features and components respectively. When richer structure is imposed on these elements, it will affect the definition of traceability relation. Then the implementation relation has to be refined to handle the resulting complexity.

References

[1] D. Benavides, S. Segura, and A. Ruiz-Corts, "Automated analysis of feature models 20 years later: a literature review," *Information Systems*, vol. 35, no. 6, pp. 615–636, 2010. [Online]. Available: http://dx.doi.org/10.1016/j.is.2010.01.001

[2] D. Beuche, H. Papajewski, and W. Schrder-Preikschat, "Variability management with feature models," *Science of Computer Programming*, vol. 53, no. 3, pp. 333 – 352, 2004, software Variability Management. [Online]. Available: http://www.sciencedirect.com/science/article/B6V17-4D04WMN-2/2/beffa7197aee601f96370977e9f25fa4

[3] K. Berg, J. Bishop, and D. Muthig, "Tracing software product line variability: from problem to solution space," in *SAICSIT '05: Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*. , Republic of South Africa: South African Institute for Computer Scientists and Information Technologists, 2005, pp. 182–191.

[4] N. Anquetil, B. Grammel, I. G. L. da Silva, J. A. R. Noppen, S. S. Khan, H. Arboleda, A. Rashid, and A. Garcia, "Traceability for model driven, software product line engineering," in *ECMDA Traceability Workshop Proceedings, Berlin, Germany*. Norway: SINTEF, June 2008, pp. 77–86.

[5] J.-M. DeBaud and K. Schmid, "A systematic approach to derive the scope of software product lines," in *ICSE '99: Proceedings of the 21st international conference on Software engineering*. New York, NY, USA: ACM, 1999, pp. 34–43.

[6] T. Eisenbarth, R. Koschke, and D. Simon, "A formal method for the analysis of product maps," in *Requirements Engineering for Product Lines Workshop, Essen, Germany*, 2002.

[7] C. Zhu, Y. Lee, W. Zhao, and J. Zhang, "A feature oriented approach to mapping from domain requirements to product line architecture," in *Software Engineering Research and Practice*, H. R. Arabnia and H. Reza, Eds. CSREA Press, 2006, pp. 219–225.

[8] T. K. Satyananda, D. Lee, S. Kang, and S. I. Hashmi, "Identifying traceability between feature model and software architecture in software product line using formal concept analysis," *Computational Science and its Applications, International Conference*, vol. 0, pp. 380–388, 2007.

[9] A. Metzger, K. Pohl, P. Heymans, P.-Y. Schobbens, and G. Saval, "Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis," in *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, 2007, pp. 243–253. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4384187

[10] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[11] E. Giunchiglia, M. Narizzano, and A. Tacchella, "Qube: A system for deciding quantified boolean formulas satisfiability," in *IJCAR*, 2001, pp. 364–369.

[12] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Formalizing cardinality-based feature models and their specialization," *Software Process: Improvement and Practice*, vol. 10, no. 1, pp. 7–29, 2005.

[13] A. V. D. Hoek, "Capturing product line architectures," in *In Proceedings of the 4th International Software Architecture Workshop*, no. CU-CS-895-99. Press, 2000, pp. 2000–95.

[14] D. S. Batory, "Feature models, grammars, and propositional formulas," in *SPLC*, ser. Lecture Notes in Computer Science, J. H. Obbink and K. Pohl, Eds., vol. 3714. Springer, 2005, pp. 7–20.

[15] BDDSolve, "http://www.win.tue.nl/ wieger/bddsolve/," 2010.